Scalability of Dynamic Storage Allocation Algorithms

Arun Iyengar
IBM Research Division
T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
aruni@watson.ibm.com

Abstract

Dynamic storage allocation has a significant impact on computer performance. A dynamic storage allocator manages space for objects whose lifetimes are not known by the system at the time of their creation. A good dynamic storage allocator should utilize storage efficiently and satisfy requests in as few instructions as possible. A dynamic storage allocator on a multiprocessor should have the ability to satisfy multiple requests concurrently. This paper examines parallel dynamic storage allocation algorithms and how performance scales with increasing numbers of processors. The highest throughputs and lowest instruction counts are achieved with multiple free list fit I. The best memory utilization is achieved using a best fit system.

1 Introduction

Many programming languages allocate storage from the *heap.* The *dynamic storage allocator,* also known as the *heap manager,* allocates and deallocates heap objects. Heap objects have indefinite lifetimes. Space occupied by a heap object is not reclaimed until the programmer or garbage collector explicitly instructs the heap manager to deallocate the object. This contrasts with frames allocated for a procedure invocation, where the space for all objects in the frame is reclaimed after the procedure terminates.

The efficiency of the dynamic storage allocator constitutes a crucial component of system performance. A good heap manager should utilize storage efficiently and satisfy requests in as few instructions as possible. A heap manager on a parallel machine should have the ability to process several requests concurrently.

This paper examines algorithms for efficiently managing heap storage on multiprocessors. Multiple free list fit I (MFLF I) [12, 11] achieves the highest throughputs and lowest total instruction counts. A best fit algorithm achieves the best memory utilization. However, the best fit algorithm has high instruction counts and low throughputs. The memory performance of multiple free list fit I is not significantly worse than best fit. When the percentage of requests for large blocks is small, the memory requirements of the two algorithms are comparable.

A number of studies on parallel dynamic storage allocation have been published. Iyengar [11, 12] studied several dynamic storage allocation algorithms and developed three new ones including MFLF I. Stone [25] presented a parallel first fit algorithm which uses the FETCH-AND-ADD instruction for concurrency control. Gottlieb and Wilson developed concurrent buddy systems which also utilize the FETCH-AND-ADD instruction [9, 10, 30]. Johnson and Davis [14] developed a parallel buddy system which coordinates allocate, deallocate, and split operations in order to minimize block fragmentation. Bigler, Allan, and Oldehoeft [2] studied three different storage allocation algorithms on a Denelcor HEP. They obtained the best performance using a parallel version of Knuth's modified first fit algorithm [15]. Ford [8] studied concurrent algorithms for real-time memory management. His algorithms use first fit. Ellis and Olson [6] studied four different first fit algorithms on a 64-node BBN Butterfly Plus. Johnson [13] has developed concurrent versions of Stephenson's fast fits algorithm for Cartesian trees [23].

2 Dynamic storage allocation algorithms

A considerable number of dynamic storage allocation algorithms have been developed [31]. Most algorithms link all free blocks together using one or more *free lists.* The simplest approach is to use one free list for all unallocated blocks. However, algorithms utilizing multiple free lists often result in better performance.

A mechanism is needed for coalescing adjacent free blocks in memory. In *immediate coalescing,* a block b is coalesced with any neighboring free blocks as soon as b is deallocated. In *deferred coalescing,* adjacent free blocks in memory are not coalesced during deallocation. The memory system operates without coalescing until a memory request cannot be satisfied. At this point, all adjacent free blocks in memory are coalesced.

Heap managers on multiprocessors should have the ability to satisfy several requests concurrently. This is typically achieved by using locking protocols which

allow a process to obtain exclusive access to free list pointers and header words of blocks. Multiple processes may search a single free list using *lock-coupling.* In lock-coupling, a process searching a list does not give up its lock on a component of a list until the process has obtained a lock on the next component. The throughput which can be obtained by a heap manager using a single free list is limited, however. More parallelism is obtained by utilizing several free lists and searching multiple free lists concurrently.

Several search strategies have been employed for systems utilizing single free lists. In *first fit,* the first block encountered which is large enough to satisfy the memory request is located. In most implementations, the search begins at the head of the free list. However, search strategies which begin at different parts of the list have also been proposed [15]. In *best fit,* the smallest block which is large enough to satisfy the memory request is located. In *worst fit,* the largest free block is located. In *random fit,* a request for a block of size n is satisfied by picking a free block of size $\geq n$ at random.

First fit and best fit systems should utilize immediate coalescing. Deferred coalescing results in many small blocks which slow down allocation requests for large blocks. Immediate coalescing can be achieved by using *address-ordered free lists.* Free blocks are ordered by memory address in an address-ordered free list. The disadvantage of address-ordered free lists is that deallocation can require a fair amount of list searching. If the free list contains n items, then deallocation requires $O(n)$ instructions.

An alternative approach is to use *boundary tags* [15]. Typically, the first word of a block is reserved for its size. Positive sizes may indicate free blocks, while negative sizes indicate allocated blocks. In the boundary tag approach, an additional tag in the last word of each block indicates whether or not the block is allocated. If the block is free, the size must also be stored at the end of the block. The free list is doubly linked to allow the deletion of items in constant time. Whenever a block is deallocated, the heap manager uses boundary tags to determine if any coalescing should take place.

Boundary tags allow blocks to be deallocated in constant time. However, extra space is required for boundary tags. The boundary tag approach requires fewer instructions but more space than the address-ordered approach.

Other approaches to allocating storage utilize multiple free lists and segregate free blocks by size. The *binary buddy system* requires all block sizes to be powers of two. Separate free lists are maintained for blocks of different sizes. Since all block sizes must be powers of two, the binary buddy system utilizes storage poorly. Other buddy systems have been proposed which use a wider variety of block sizes [11]. Buddy systems may use either immediate or deferred coalescing.

Stephenson implemented an algorithm known as fast fits [24, 23] which uses Cartesian trees [27] for storing free blocks. The structure of a Cartesian tree is determined by both a primary and a secondary key. The nodes of the tree are totally ordered with respect to the primary key and partially ordered with respect to the secondary key. An inorder traversal of the tree yields the nodes ordered by the primary key. Each node has a secondary key at least as big as any secondary key belonging to a descendant of the node. In fast fits, block addresses are used as primary keys while block sizes are used as secondary keys. One of the disadvantages of Cartesian trees is that they can become unbalanced. In the worst case, search times will be $O(n)$ instead of $O(log(n))$ where n is the number of free blocks.

Quick fit [28, 29] uses *quick lists* for the most frequently requested sizes. Allocation from a quick list requires few instructions. A quick list exists for each block size s defined over the interval

$$minQL \leq s \leq maxQL.$$

We will assume that $minQL$ is the minimum legal block size. The optimal value for $maxQL$ depends on the request distribution. A *small block* is a block of size $\leq maxQL$. A *large block* is a block of size $> maxQL$. A single miscellaneous list, or *misc list,* exists for large free blocks. Quick fit utilizes deferred coalescing. Figure 1 shows how quick fit and MFLF I organize free storage for small blocks.

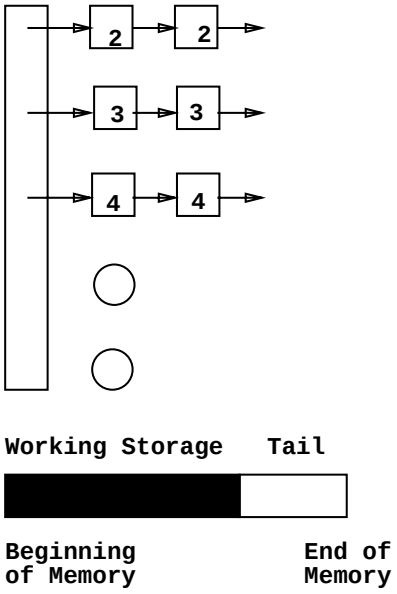

Figure 1: Quick fit and MFLF I both use multiple free lists known as *quick lists* to segregate free blocks according to their sizes. In this figure, the number in each block represents its size. Contiguous storage at one end of memory which has not been allocated since memory was last coalesced is known as the *tail.* Allocation from quick lists and the tail requires few instructions.

Memory is divided into the *tail* and *working storage.* The tail is a contiguous block of free words at one

end of memory which has not been allocated since memory was last coalesced. Working storage consists of memory which is not part of the tail (Figure 1). Initially, the tail constitutes the entire heap memory, and each free list is empty. Blocks are added to free lists during deallocations.

Quick fit has a number of desirable characteristics. Allocation from quick lists is fast; the vast majority of requests can usually be satisfied from quick lists. Deallocation is always fast; newly freed blocks are simply placed at the beginning of the appropriate free list. Quick fit has a high degree of parallelism because different free lists may be examined concurrently.

A significant number of instructions may be required to allocate a large block. However, most requests on real memory systems are for small blocks. Therefore, the average number of instructions required to satisfy heap requests is usually low. The instruction count tends to increase with the fraction of requests for large blocks.

2.1 Multiple free list fit I

Multiple free list fit I uses the same approach as quick fit for allocating small blocks and one or more misc lists for large blocks. By contrast, quick fit always uses a single misc list for storing large free blocks. Good performance is achieved by using about five misc lists. Each large free block is placed on a misc list according to its size. The allocation algorithm for small blocks is similar to the algorithm used by quick fit. We say that a quick fit and MFLF I allocator are *congruent* if they both use identical sets of sizes for quick lists.

Misc lists are arranged by block size. Suppose the system has n misc lists designated by l_1 through l_n. Let max be the size of the largest block which can ever exist in the system, low_i the size of the smallest block which can be stored on l_i, and $high_i$ the size of the largest block which can be stored on l_i. We refer to the pair $(low_i, high_i)$ as a *misc list range.* Misc lists cover disjoint size ranges. If

$$1 \leq i < n,$$

then

$$high_i + 1 = low_{i+1}.$$

The boundary conditions are

$$low_1 = maxQL + 1$$

and

$$high_n = max.$$

Whenever a process accesses one or more misc lists in order to obtain a block of size s, MFLF I attempts to satisfy the request without performing a first fit search of a misc list. First fit searches can be time consuming. By contrast, if a nonempty misc list l_i exists such that $low_i \geq s$, the first free block contained on l_i can be used to satisfy the request.

2.1.1 Allocating small blocks

The following strategy is used to allocate a block of size s where $s \leq maxQL$:

1. If the quick list for blocks of size s is nonempty, allocate the first block from the list.
2. If the previous step fails, try to satisfy the request from the tail.
3. If the previous step fails, examine lists containing larger blocks until a free block is found. This search is conducted in ascending block size order beginning with the list storing blocks of size $s+1$.
4. If the previous step fails, coalesce all adjacent free blocks in memory and go to Step 1.

2.1.2 Allocating large blocks

The following strategy is used for allocating a block of size s where $low_n \geq s > maxQL$:

1. Locate misc list l_i, where l_i is the list with the smallest value of low_i such that $low_i \geq s$.
2. Examine lists l_i through l_n in ascending block size order until a free block is found. As soon as a nonempty list is located, the first block contained on the list is used to satisfy the request.
3. If the previous step fails and $low_i > s$, then examine list $i-1$ using first fit. If $low_i = s$, go to the next step.
4. If the previous step fails, try to satisfy the request from the tail.
5. If the previous step fails, coalesce all adjacent free blocks in memory and go to Step 1.

The following strategy is used for allocating a block of size s where $s > low_n$:

1. Search list l_n using first fit.
2. If the previous step fails, try to satisfy the request from the tail.
3. If the previous step fails, coalesce all adjacent free blocks in memory and go to Step 1.

An example of allocation using MFLF I is shown in Figure 2.

2.1.3 Deallocating blocks

In order to deallocate a block of size $s \leq maxQL$, the block is placed at the head of the appropriate quick list. In order to deallocate a block of size $s > maxQL$, the block is placed at the head of misc list i, where

$$low_i \leq s \leq high_i.$$

Since deferred coalescing is used, adjacent free blocks are not coalesced during a deallocation.

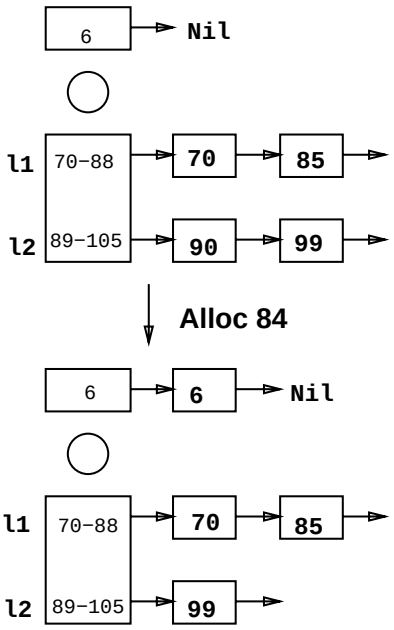

Figure 2: Satisfying a request for a large block using multiple free list fit I. List l2 is examined before l1. Since the smallest block on l2 must contain at least 89 words, the first block on l2 will always be large enough to satisfy a request of size 84. By contrast, satisfying the request from l1 would have required searching the list.

2.1.4 Discussion

Different misc lists may be searched concurrently. Multiple processes may search the same misc list via lock-coupling, just as in quick fit. However, MFLF I often obtains significantly more parallelism than quick fit because of multiple misc lists. If a single misc list is used, a small number of processes can lock parts of the list and slow down other allocating processes. Elements at the front of the list tend to become bottlenecks. Contention for these list elements can substantially increase the time required for allocating large blocks.

Multiple misc lists in an MFLF I system reduce the contention on elements at the beginning of lists. Misc list ranges are selected in order to distribute requests as evenly as possible across different misc lists. We have encountered many situations where an MFLF I system with multiple misc lists has substantially more parallelism than a congruent quick fit system.

In order to manage large blocks, a data structure d associating each misc list range with a free list pointer is searched. One optimization which can reduce this searching is to store free list pointers in an array indexed by block sizes. Some searching of d may still be required to prevent the array from becoming too large.

It is common for a program to repeatedly allocate and deallocate several blocks of the same size, s. If deferred coalescing is used, free blocks of size s will frequently exist in the system. Let r be a request for a block of size s. A dynamic storage allocator satisfies request r with an *exact fit* if the block satisfying the request contains exactly s words without being split. Exact fits are desirable because they result in good memory utilization.

Multiple free list fit I always tries to satisfy requests for small blocks with exact fits. However, MFLF I often does a poor job of finding exact fits for large blocks. The probability of finding exact fits is increased by choosing appropriate misc list ranges. If large blocks of size s are frequently allocated and deallocated, then misc list ranges should be chosen so that

$$s = low_i$$

for some misc list l_i. A search for a block of size s will then examine l_i before examining any other list.

An alternative strategy which does not rely on misc list range values is to modify the search strategy for large blocks. The first list examined during a search for a block of size s is list l_i, where

$$low_i \leq s \leq high_i.$$

The drawback to this approach is that if $s > low_i$, then a first fit search of l_i takes place. Allocation is thus slightly slower on average.

We have experimented with versions of MFLF I which dynamically adjust themselves to request distributions. Misc list ranges are periodically recalculated in order to match request distributions [11]. This approach was not successful. The overhead for maintaining information about request distributions and recalculating misc list ranges more than compensated for other performance gains.

3 Performance comparisons

3.1 Methodology

The criteria we use for comparing dynamic storage algorithms are total instructions executed, throughput, and memory utilization. The throughput of a dynamic storage allocator is the average number of requests which can be satisfied in 1000 cycles.

Memory is wasted due to *internal fragmentation* and *external fragmentation.* Internal fragmentation is memory wasted by satisfying a request with a larger block than is needed. We define internal fragmentation at any time t as:

$$f_i = a/b, \tag{1}$$

where a is the total number of words allocated at time t and b is the number of words which would have been allocated in a system with no internal fragmentation. The aggregate internal fragmentation calculated from l measurements is given by the formula:

$$\overline{f_i} = \frac{\sum_{j=1}^{l} a_j}{\sum_{j=1}^{l} b_j}, \tag{2}$$

where a_j is the number of allocated words at the time of measurement j and b_j is the number of words which would have been allocated in a system with no internal fragmentation.

The dynamic storage allocators presented in this paper require one extra word per block for storing the block size. This extra word per block is not considered to be storage wasted by internal fragmentation. However, the first fit system utilizing boundary tags uses two extra words per block. One of the extra words is considered to result from internal fragmentation.

We attempt to split blocks during allocation to minimize internal fragmentation in every allocator except for the buddy system. Let min_block_size be the minimum block size allowed by a dynamic storage allocator. Whenever a block of size s is used to satisfy a request for r words and $s \geq r + min\_block\_size$, the block is split and a fragment of size $s - r$ is returned to free storage in every allocator except for the buddy system.

External fragmentation occurs when free blocks are interleaved with allocated blocks. If m is the maximum number of consecutive free words of memory, then a request for a block of size $> m$ cannot be satisfied, even if the total number of free words is substantially larger than m. External fragmentation is determined by running the heap manager until an allocation request cannot be satisfied. External fragmentation is given by the formula:

$$f_e = M/a, \tag{3}$$

where M is the number of words in memory and a is the number of allocated words when memory overflows. The aggregate external fragmentation calculated from n experiments is given by the formula:

$$\overline{f_e} = \frac{\sum_{j=1}^{n} M_j}{\sum_{j=1}^{n} a_j}, \tag{4}$$

where M_j is the memory size and a_j is the number of allocated words when memory overflows in experiment j.

Total fragmentation is memory lost to either internal or external fragmentation. We define total fragmentation quantitatively as the product of internal and external fragmentation:

$$f_t = f_i * f_e. \tag{5}$$

Aggregate total fragmentation calculated from several experiments is given by the formula:

$$\overline{f_t} = \overline{f_i} * \overline{f_e}. \tag{6}$$

A dynamic storage allocator which doesn't waste any memory would have a total fragmentation value of 1. A dynamic storage allocator which requires 10% more storage than an allocator which doesn't waste any storage would have a total fragmentation value of 1.1.

Our performance results were obtained using the Id World programming environment [18]. Id World simulates a shared memory parallel machine where the number of processors can be varied from one to infinity. The simulated machine has the following characteristics:

- Each operator takes unit time to execute.
- Results of an instruction are available to its successors instantaneously. No communication delay exists.
- All enabled instructions are executed immediately.

Each processor is sequential. Thus, the throughput achieved by one processor is a measure of the instruction count one would expect to see on a purely sequential machine.

Experiments consisted of several loop iterations. During each iteration, 50 blocks were allocated concurrently. Block lifetimes were distributed exponentially with a mean of 3 loop iterations.

One way to increase the throughput of a storage allocator is to divide the heap into several different areas and distribute requests to the different areas. For example, the throughput of a first fit system can be increased by using several free lists instead of a single one. This multiplicity introduces complexity, however. A method is needed to distribute requests to different free lists.

For machines with many processors, dividing the heap into different regions becomes a necessity. Algorithms sustaining higher throughputs will require fewer subdivisions of the heap for good performance. For example, MFLF I, quick fit, and a first fit system using boundary tags have been implemented on the Monsoon multiprocessor [20, 26] to support the execution of parallel programs. Each processor allocates storage from a different area of the heap. Consequently, the throughput of the heap allocator scales with the number of processors. Since processors are pipelined, a single processor may have several pending memory requests at the same time. Multiple free list fit I and quick fit generally satisfy requests without bottlenecking the system. By contrast, the first fit system using boundary tags often bottlenecks the system because only one process is allowed to access a free list at a time.

3.2 Results

The six algorithms described in Table 1 were compared using three different request distributions. Distribution I consisted of blocks whose sizes conformed to an exponential distribution truncated at 30. The mean of the distribution was 10. Of the blocks in Distribution II, 50% conformed to Distribution I. The remaining 50% were uniformly distributed over the closed interval between 31 and 230. Of the blocks in Distribution III, 50% conformed to Distribution I. The remaining 50% were uniformly distributed over the sizes 40, 50, 65, 85, 100, 120, 150, 200, 250, and 300. For MFLF I and quick fit, quick lists were used for all blocks containing up to 30 words.

Most programs only request a small percentage of large blocks. Distribution I is a reasonably good approximation of the average case. Most published request distributions are similar to Distribution I. Distribution III is probably more representative of a single program requesting a high percentage of large blocks than Distribution II because large blocks requested by single programs tend to be distributed over a small number of block sizes.

The average number of instructions required by the algorithms for satisfying a request is shown in Figure 3. The throughputs of the six algorithms are shown in Figures 4, 5, 6, and 7. Memory utilization is shown in Figures 8, 9, and 10.

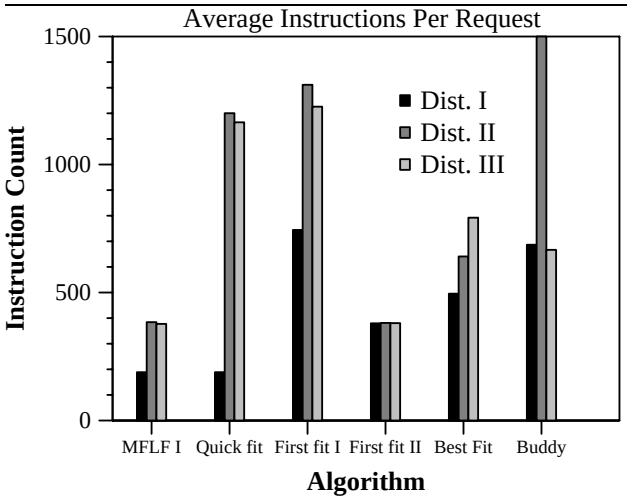

Figure 3: The average number of instructions for satisfying a request. The buddy system actually requires an average of 2594 instructions for satisfying a request on Distribution II.

3.3 Discussion

MFLF I generally results in the highest throughputs. On Distribution I, quick fit and MLFL I give the same performance because the two algorithms use the same approach for allocating small blocks. MFLF I achieves considerable higher throughputs than quick fit on Distributions II and III where the percentage of requests for large blocks is significant.

MFLF I and quick fit have considerably lower instruction counts than the other algorithms for Distribution I. For Distributions II and III, MFLF I and the first fit system using boundary tags have the lowest instruction counts. Unfortunately, the first fit system using boundary tags doesn't scale well. In a parallel environment, the comparative performance advantage of MFLF I over the first fit system increases significantly.

However, the first fit system using boundary tags (Algorithm 4) has a couple of advantages over MFLF I. One advantage is that Algorithm 4 is slightly easier to

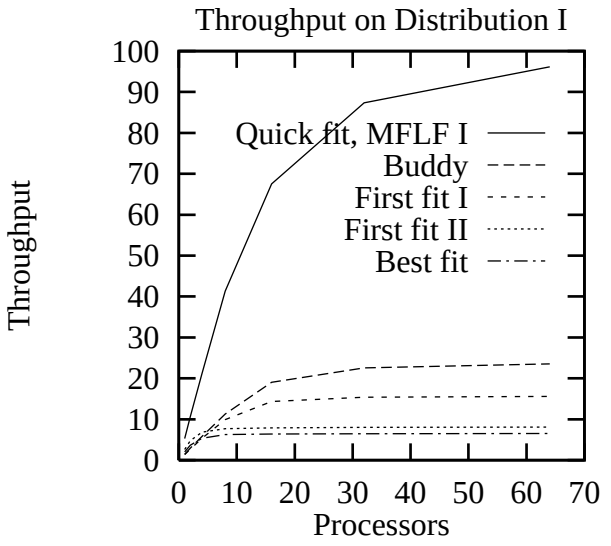

Figure 4: The throughputs of the six algorithms on Distribution I.

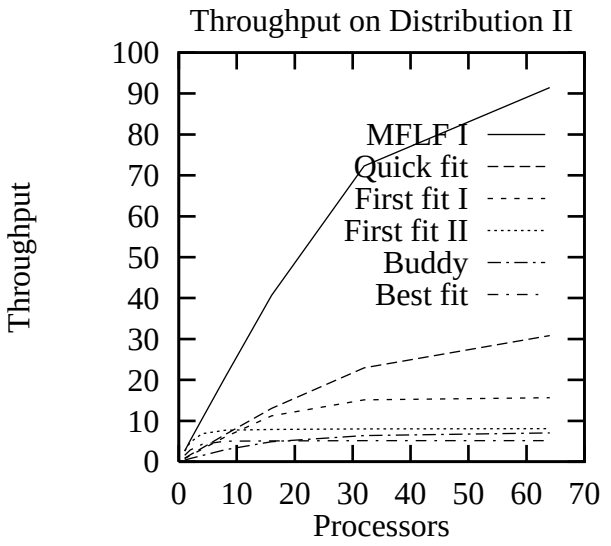

Figure 5: The throughputs of the six algorithms on Distribution II.

Algorithm	*Coalescing Strategy*	*Free List Structure*	*Multiple Free Lists?*	*Lock-Coupling?*	*Minimum Block Size*
1 (MFLF I)	Deferred	Singly Linked	Yes	Yes	2
2 (Quick Fit)	Deferred	Singly Linked	Yes	Yes	2
3 (First Fit I)	Immediate	Address-ordered, Singly Linked	No	Yes	2
4 (First Fit II) (boundary tags)	Immediate	Unordered, Doubly Linked	No	No	4
5 (Best Fit)	Immediate	Address-ordered, Singly Linked	No	No	2
6 (Buddy)	Immediate	Doubly Linked	Yes	No	4

Table 1: This table summarizes the characteristics of the algorithms which we tested. Algorithms with multiple free lists allow different lists to be examined concurrently. If a free list does not support lock-coupling, only one process may examine the list at a time.

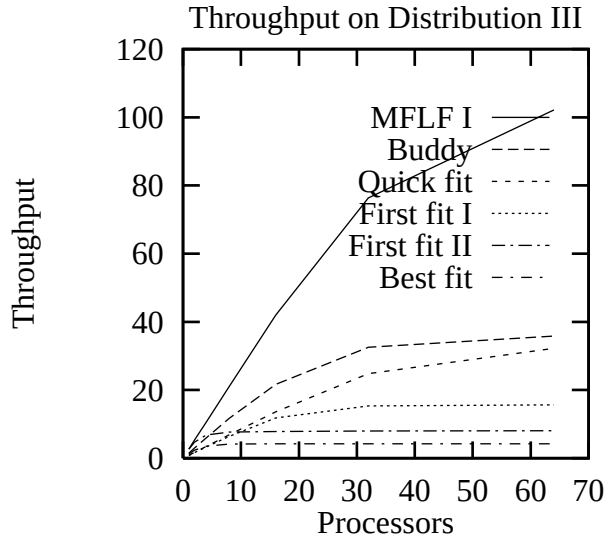

Figure 6: The throughputs of the six algorithms on Distribution III.

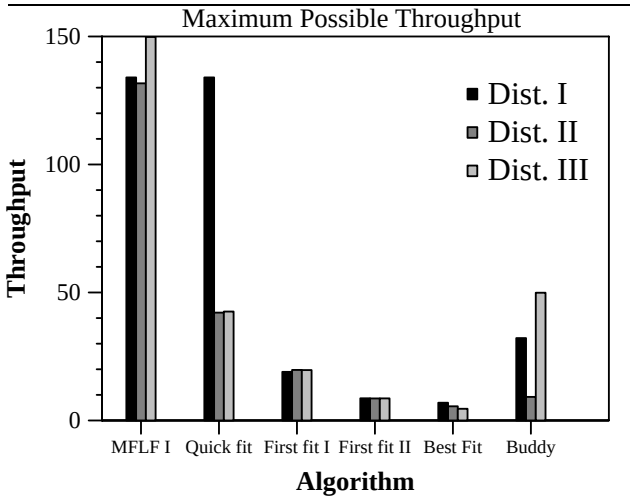

Figure 7: The maximum possible throughputs of the algorithms. This was obtained by simulating a machine with an unbounded number of processors.

implement than MFLF I. Another advantage is that the performance of Algorithm 4 varies less than the performance of any of the other algorithms. Algorithm 4 would be a good algorithm to use when one wants a dynamic storage allocator with little variation in response times.

Since quick fit and MFLF I use deferred coalescing, an allocation request can take a long time to satisfy if the request triggers the coalescer. This can be bad for a real-time system requiring predictable response times. When memory is sufficiently large, however, coalescing rarely happens and the average number of instructions to satisfy a request is not substantially affected by the coalescer. When memory is almost full, coalescing happens more frequently and the overhead can be significant. We didn't encounter high coalescing frequencies until memory was near the overflow point.

The binary buddy system can sustain moderately high throughputs due to the presence of multiple free lists. The major drawback is that memory is used inefficiently. About 40% of memory is wasted because block sizes have to be rounded up to the nearest power of two. The binary buddy system is generally the fastest buddy system. Other buddy systems have been proposed which use a wider variety of block sizes in order to reduce internal fragmentation [11]. However, there is a tradeoff between internal and external fragmentation [21, 22, 4]. Buddy systems which

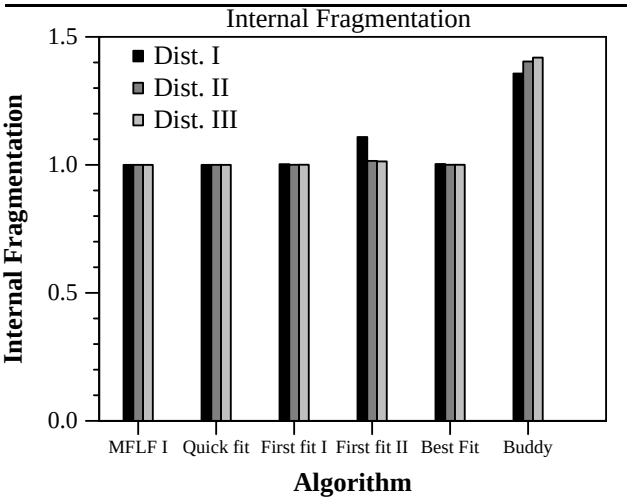

Figure 8: Storage wasted due to internal fragmentation. The buddy system wastes the most storage because all block sizes have to be rounded up to the nearest power of two.

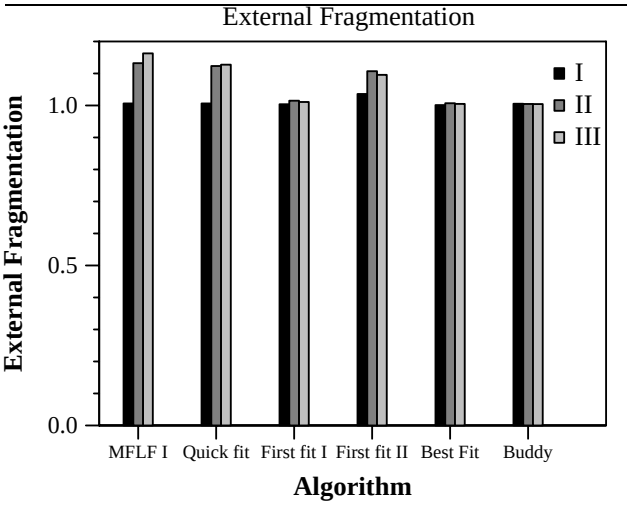

Figure 9: Storage wasted due to external fragmentation.

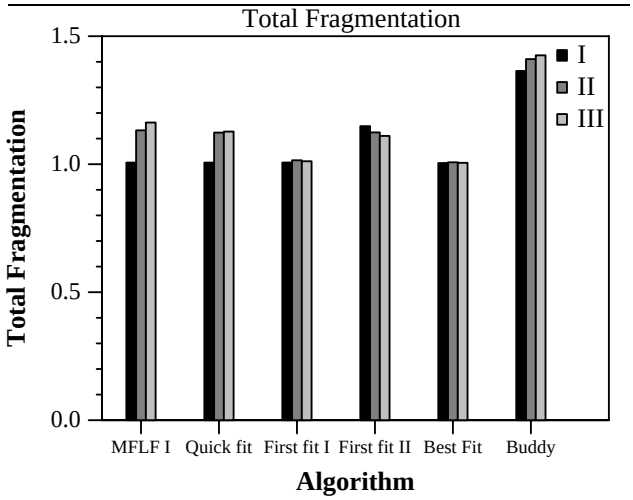

Figure 10: Total storage overhead. A memory system which wastes no storage would have a value of 1.0. A memory system which requires 10% more storage than a system which wastes no storage would have a value of 1.1. Best fit achieves the best memory utilization followed by first fit I.

produce less internal fragmentation than the binary buddy system tend to produce more external fragmentation. The weighted ss buddy system [22] and the dual buddy system [19] have been touted as displaying less total fragmentation than the binary buddy system for certain distributions. However, the weighted ss buddy system is significantly slower than the binary buddy system. Furthermore, both the weighted ss and dual buddy systems still display a substantial amount of fragmentation. If memory utilization is critical, a storage allocator other than the buddy system should be used.

The best fit system using address-ordered free lists (Algorithm 5) achieves the best memory utilization. This algorithm is slow and sequential, however. Our results agree with others which found that best fit wastes less memory than first fit [5, 7, 28, 17, 16, 1, 23, 3]. Knuth is one of the few who concluded that first fit results in better memory utilization than best fit [15].

The first fit system using address-ordered free lists (Algorithm 3) results in the second best memory utilization. Its memory utilization is similar to that of MFLF I and quick fit on Distribution I and superior to the other two algorithms on Distributions II and III. The first fit system using boundary tags (Algorithm 4) wastes more memory than Algorithm 3 because an extra word is required for each block and minimum block sizes are four words instead of two. This overhead is accounted for as internal fragmentation and decreases with increasing block size. When a high percentage of very small blocks are requested, the space overhead from Algorithm 4 may be significant.

MFLF I and quick fit achieve good space utiliza-

tion when the percentage of requests for large blocks is small as is usually the case. Memory utilization degrades as the percentage of requests for large blocks increases. We have obtained high throughputs and good memory utilization using an algorithm known as Multiple Free List Fit II (MFLF II) [12, 11]. MFLF II approximates best fit and has throughputs comparable to MFLF I. MFLF II is identical to MFLF I and quick fit when all requests are for small blocks as in Distribution I. On Distributions II and III, MFLF II wastes more memory than Algorithms 5 and 3 but less memory than any of the other algorithms compared in this study. MFLF II is harder to implement than any of the other algorithms.

The variability in speed is significantly higher than the variability in memory utilization. The only algorithms which waste significant amounts of storage are the buddy system and the first fit system using boundary tags when request distributions are skewed to smaller block sizes than Distribution I. If memory overheads of around 15% are tolerable, then Algorithms 1, 2, 3, and 5 are adequate. Since MFLF I is the fastest algorithm, it is our first choice.

Request distributions can have a significant affect on the performance of dynamic storage allocators. The ultimate test is how the allocators perform on real request distributions. A number of request distributions from real systems have been published [11]. For most of these distributions, the vast majority of requests are for blocks smaller than 30. Many of these distributions can be approximated by an exponential distribution similar to Distribution I. The problem with published distributions is that they tend to be averaged over a large number of programs. Individual programs show huge variations in request patterns. We have simulated dynamic storage allocators on published distributions and the results tend to be similar to the results produced by Distribution I. In practice, however, some programs request a high percentage of large blocks and storage allocators need to handle these cases efficiently.

We have implemented MFLF I, quick fit, and a first fit system using boundary tags (Algorithm 4) on the Monsoon multiprocessor [20, 26]. For most applications, the performance of MFLF I and quick fit are similar. Algorithm 4 generally requires about twice as many instructions as the other two and has significantly less parallelism. On programs requesting a significant percentage of large blocks, MFLF I generally has higher throughputs and lower instruction counts than the other two algorithms. In some cases, quick fit requires over ten times as many instructions as MFLF I.

4 Summary and conclusion

This paper has compared several dynamic storage allocation algorithms. We evaluated the algorithms in terms of instruction counts, throughput, and memory utilization. Multiple free list fit I (MFLF I) generally results in the highest throughputs and lowest instruction counts. When the percentage of requests for large blocks is small as is usually the case, quick fit and MFLF I result in the lowest instruction counts and highest throughputs. The performance of quick fit degrades considerably as the percentage of requests for large blocks increases. When the percentage of large blocks is high, MFLF I achieves significantly higher throughputs than the other algorithms. MFLF I and a first fit system using boundary tags achieve the lowest instruction counts under this scenario.

Memory utilization is maximized by a best fit system using address-ordered free lists. This algorithm is slow and sequential, however. A first fit using address-ordered free lists makes the second best use of memory. The variability in memory utilization is significantly less than the variability in instruction counts and throughputs. MFLF I generally wastes less than 20% of memory. We thus recommend MFLF I as the overall best algorithm in terms of total instructions, throughput, and memory utilization.

References

[1] L. L. Beck. A Dynamic Storage Allocation Technique Based on Memory Residence Time. *Communications of the ACM*, 25(10):714–724, October 1982.

[2] B. M. Bigler, S. J. Allan, and R. R. Oldehoeft. Parallel Dynamic Storage Allocation. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 272–275, 1985.

[3] G. Bozman et al. Analysis of Free-Storage Algorithms - Revisited. *IBM Systems Journal*, 23(1):44–64, 1984.

[4] A. G. Bromley. Memory Fragmentation in Buddy Methods for Dynamic Storage Allocation. *Acta Informatica*, 14:107–117, 1980.

[5] G. O. Collins. Experience in Automatic Storage Allocation. *Communications of the ACM*, 4(10):436–440, October 1961.

[6] C. S. Ellis and T. J. Olson. Algorithms for Parallel Memory Allocation. *International Journal of Parallel Programming*, 17(4):303–345, 1988.

[7] J. S. Fenton and D. W. Payne. Dynamic Storage Allocation of Arbitrary Sized Segments. In *Proceedings of IFIPS*, pages 344–348, 1974.

[8] R. Ford. Concurrent Algorithms for Real-Time Memory Management. *IEEE Software*, pages 10–23, September 1988.

[9] A. Gottlieb and J. Wilson. Using the Buddy System for Concurrent Memory Allocation. Technical Report Ultracomputer System Software Note 6, Courant Institute, New York University, 1981.

[10] A. Gottlieb and J. Wilson. Parallelizing the Usual Buddy Algorithm. Technical Report Ultracomputer System Software Note 37, Courant Institute, New York University, 1982.

[11] A. K. Iyengar. Dynamic Storage Allocation on a Multiprocessor. Technical Report MIT/LCS/TR-560, MIT Laboratory for Computer Science, Cambridge, MA, December 1992. PhD Thesis.

[12] A. K. Iyengar. Parallel Dynamic Storage Allocation Algorithms. In *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 82–91, December 1993.

[13] T. Johnson. A Concurrent Fast-Fits Memory Manager. Technical Report TR91-009, University of Florida Department of CIS, September 1991.

[14] T. Johnson and T. Davis. Parallel Buddy Memory Management. *Parallel Processing Letters*, 2(4):391–398, 1992.

[15] D. E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Reading, MA, second edition, 1973.

[16] C. H. C. Leung. An Improved Optimal-Fit Procedure for Dynamic Storage Allocation. *The Computer Journal*, 25(2):199–206, 1982.

[17] N. R. Nielsen. Dynamic Memory Allocation in Computer Simulation. *Communications of the ACM*, 20(11):864–873, November 1977.

[18] R. S. Nikhil et al. Id World Reference Manual. Technical report, MIT Laboratory for Computer Science, Cambridge, MA, November 1989.

[19] I. P. Page and J. Hagins. Improving the Performance of Buddy Systems. *IEEE Transactions on Computers*, C-35(5):441–447, May 1986.

[20] Gregory M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report MIT/LCS/TR-432, MIT Laboratory for Computer Science, Cambridge, MA, 1988. PhD Thesis.

[21] J. L. Peterson and T. A. Norman. Buddy Systems. *Communications of the ACM*, 20(6):421–431, June 1977.

[22] N. M. Pitman, F. W. Burton, and E. W. Haddon. Buddy Systems with Selective Splitting. *The Computer Journal*, 29(2):127–134, 1986.

[23] C. J. Stephenson. Fast Fits: New Methods for Dynamic Storage Allocation. Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1983.

[24] C. J. Stephenson. Fast Fits: New Methods for Dynamic Storage Allocation. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 30– 32, 1983.

[25] H. S. Stone. Parallel Memory Allocation using the FETCH-AND-ADD Instruction. Technical Report RC 9674, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, November 1982.

[26] K. R. Traub, G. M. Papadopoulos, M. J. Beckerle, J. E. Hicks, and J. Young. Overview of the Monsoon Project. In *Proceedings of the 1991 IEEE International Conference on Computer Design*, October 1991.

[27] J. Vuillemin. A Unifying Look at Data Structures. *Communications of the ACM*, 23(4):229–239, April 1980.

[28] C. B. Weinstock. *Dynamic Storage Allocation Techniques*. PhD thesis, Carnegie-Mellon University, 1976.

[29] C. B. Weinstock and W. A. Wulf. Quick Fit: An Efficient Algorithm for Heap Storage Allocation. *SIGPLAN Notices*, 23(10):141–148, 1988.

[30] J. Wilson. *Operating System Data Structures for Shared-Memory MIMD Machines with Fetch-and-Add*. PhD thesis, New York University, 1988.

[31] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the Memory Management International Workshop IWMM 95*, pages 1–126, September 1995.